Design and Experience: Using the Intel® Itanium® 2 Processor Performance Monitoring Unit to Implement Feedback Optimizations

Youngsoo Choi Allan Knies Geetha Vedaraman Jeremiah Williamson

Itanium® Architecture and Performance Team
Intel Corporation
2200 Mission College Blvd
Santa Clara, CA
{youngsoo.choi, allan.knies, geetha.vedaraman, jeremiah.d.williamson}@intel.com

Abstract

Historically, profile-guided optimization has gathered its profile data by executing an instrumented binary and capturing the output. While this approach enables the collection of function and basic block frequencies, it cannot extract microarchitectural event information such as cache activity, TLB activity, and branch prediction behavior. Using instrumentation also requires that programs be compiled with different options (one for the profile run, one for the optimization run) with the profiling run taking substantially longer due to instrumentation overhead and reductions in compiler optimization. To help address these issues, the Intel® Itanium® 2 processor has extensive hardware support to allow for highly accurate instruction-specific information to be gathered from any binary. In this paper, we cover three broad topics: the Itanium® 2 processor performance monitoring unit (PMU), our tools and methodology to gather and process cache, TLB, and branch activity information, and a case study where we demonstrate the entire system to reduce data access stalls.

Keywords: Profile feedback, profile-guided, feedback-directed, optimization, Itanium® architecture, Intel® Itanium®2 processor, performance monitoring, compiler optimization, data prefetching

1 Introduction

Profile-guided optimization (PGO) has historically been an important technology for improving the performance of applications. On the Intel® Itanium® 2 processor, current compilers achieve a 10% to 20% performance improvement using instrumentation-based edge profiling PGO. Although PGO provides performance benefits, there are several barriers that prevent its universal usage. Most practically, PGO requires an extra step in the code development process to perform profiling runs and process the data.

Various research groups have developed systems to reduce the cost of gathering profile data [BDB00][CL99][MTB+01][PL01], but these are generally based on edge or path profiles rather than full microarchitectural monitoring. While our system still requires a second compilation step, we have dramatically reduced the overhead while increasing the amount, detail, and quality of data provided. In Section 2, we will describe the Itanium®2 processor performance monitoring features, our infrastructure for exploiting it, and measurements of overhead for gathering it.

Other systems have allowed full-time monitoring of real binaries in conjunction with re-optimization [And+97], but such approaches were generally deployed with instruction pointer (IP) sampling and extensive off-line processing to reconstruct the data into a usable form. More recent systems, such as the Alpha 21264 and Pentium 4 event based sampling systems have support for obtaining detailed microarchitectural info [Spr02] [DHW+96], but we are not aware of any current compilers that use microarchitectural feedback data. Our compiler is the first to enable program optimization based on a processor that has dedicated support for gathering detailed instruction-specific branch, cache, and TLB behavior. In Section 3, we discuss our initial experience from applying data cache access information to optimize cache behavior using an internal version of the Intel product compiler.

2 Description of the Monitors and Framework

2.1 Intel® Itanium® 2 Processor Performance Monitor Unit (PMU)

The Itanium® architecture performance monitors are designed to help characterize and optimize the behavior of applications using on- or off-line tools. The Itanium®2 PMU counters are 48-bits wide and can count as many as four microarchitectural events simultaneously (out of about 150 possible events) and allow separate or

combined monitoring of OS and application code. In addition to the basic events, the PMU provides a variety of filters (also called qualifiers) that allow finer control of counters. Available filters include an opcode matcher, instruction and data address range matchers, and a privilege level matcher. For instance, by writing the opcode pattern for loads into the PMU's opcode mask register, the instructions retired counter will only count those instructions that are loads. In general, this provides the ability to count different types of instructions using just one counter. These filtering capabilities amplify the usefulness of event counters without requiring substantially more hardware. Table 1 shows the results of using opcode matching to measure instruction mixes on the Itanium® 2 processor for the CPU2000 integer benchmarks.

	ALU	Nop	BR	FP	LD	ST	Other
164.gzip	39%	25%	10%	0%	14%	3%	9%
175.vpr	31%	24%	8%	5%	14%	5%	14%
176.gcc	38%	18%	10%	0%	14%	10%	11%
181.mcf	38%	21%	9%	0%	15%	6%	10%
186.crafty	45%	17%	9%	0%	16%	2%	11%
197.parser	36%	25%	11%	0%	15%	3%	9%
252.eon	25%	31%	4%	12%	12%	9%	8%
253.perlbmk	36%	20%	12%	0%	15%	3%	13%
254.gap	35%	19%	9%	0%	22%	5%	10%
255.vortex	39%	14%	13%	0%	15%	7%	12%
256.bzip2	40%	16%	11%	0%	15%	7%	11%
300.twolf	29%	26%	6%	5%	9%	3%	21%

Table 1 Instruction breakdown

While the Itanium®2 PMU has a large set of events and filtering capabilities, it also provides a set of *cycle accounting* events that identify and classify how time is being spent in the program. During execution, cycle accounting hardware categorizes each execution cycle into bins that represent different types of stalls. The following is a high level description of these bins:

? front-end pipeline (Icache misses)
? register stack engine (register spill/fill traffic)
? branch mispredictions and exceptions
? back-end pipeline stalls
? integer execution stalls due to register dependency
? FP execution stalls due to register dependency
? Load to use stalls (on cache misses)

Any cycles not binned into one of these categories are 'unstalled cycles' and represent times when the pipeline is flowing freely. Cycle accounting is organized hierarchically so a user can do a high level binning with the top-level categories and then drill down into specific trouble areas by fine-tuning the event parameters. Figure 1 shows the cycle accounting breakdown for the Itanium® 2 processor running the CPU2000 integer benchmarks.

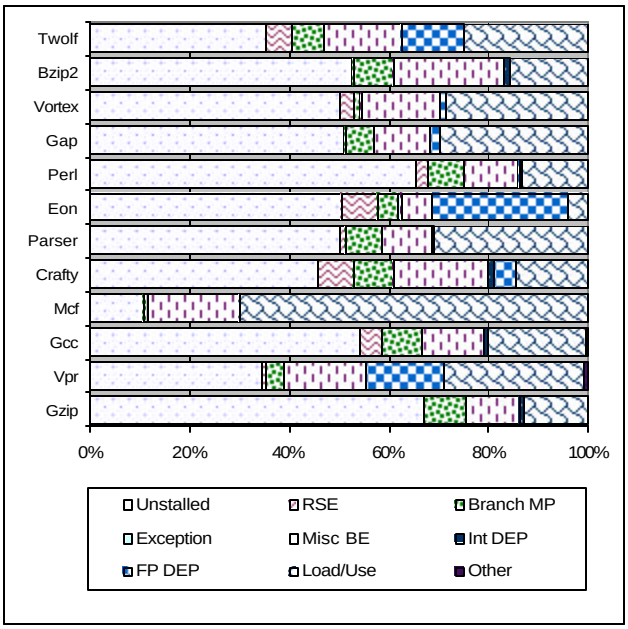

Figure 1 Cycle accounting for CPU2k Integer

While the features described so far are very valuable in characterizing the overall behavior of applications, for feedback optimization, we are interested in event data that is linked to specific locations in the program. To do this, the Itanium architecture provides event address registers (EARs) that can be programmed to capture microarchitectural event information and the instruction pointer (IP) which caused the event. There are EARs for data, instruction, and branch events. Each of these EARs captures the IP plus EAR-specific event information.

The data EAR can capture events related to cache, TLB, and ALAT misses. When programmed to track data cache misses, the processor samples load misses at a programmable frequency and records the instruction address, the data address, and the latency needed to access the data. When programmed to monitor TLB activity, the data EAR captures the level in the hierarchy that services the translation (first or second level TLB, hardware page walker, or the operating system). When programmed to capture ALAT information, only the IP of chk.a or ld.c that miss in the ALAT are recorded.

For branches, the 'EAR' is actually an eight entry branch trace buffer (BtrB) that continuously captures the IP and target address or prediction information of branches. The BtrB can be programmed to record all branches or specific subsets of branches (only taken or not-taken branches, only branches that correctly or incorrectly predict the target or direction, or only for

branches that are of specific types such as call, return, loop-type, etc.). The BtrB continuously gathers events until a programmed threshold of collected events is reached (e.g., X mispredicts, or Y branches executed, etc.)

Using these capabilities, the BtrB can be used to replace instrumentation-based edge profiling with statistical edge/block profiling by sampling the BtrB and mathematically reconstructing approximate edge weights for the graph. Other groups at Intel have implemented such systems [BN00] and have demonstrated that the statistical sampling of the BtrB yielded profiles comparable to instrumentation-based approaches.

The BtrB can also be used to identify the branches that mispredict most frequently by programming the buffer to only record those branches that mispredict. The mispredicted branches are then statistically sampled from the BtrB. This section has provided just a brief introduction to the capabilities of the Itanium® 2 PMU – please see [Int02] for full details.

2.2 Programming, Sampling, and Storing PMUs

To program and control the PMU, we use a tool called *psamp*, which is part of Intel's Vtune™ Performance Analyzer[1][IntV02]. *Psamp* takes command line inputs and writes the performance monitor control (PMC) registers to set the sampling rates, filtering options, events to count, etc. *Psamp* gathers EAR samples in an internal buffer until it fills up or the monitored application terminates, at which point the samples are written to disk in a binary format called .TB3.

The overhead for collecting the EAR samples is relatively small and configurable. Table 2 shows the overhead for enabling sampling using the SPEC CPU2000 integer benchmarks as the workload and measuring the impact of different sampling rates.

Event on which sampling is based	Sampling rate					
	1/1M	1/500k	1/100k	1/50k	1/10k	1/1k
DATA_EAR_EVENTS (Cache Miss)			0.19%	0.28%	0.89%	5.33%
DATA_EAR_EVENTS (TLB Miss)			0.14%	0.14%	0.27%	1.53%
BRANCH_EVENT	0.44%	0.86%	3.45%	6.85%		
CPU_CYCLES	1.00%	2.23%	8.80%	15.98%		

Table 2 Sampling overhead as a percentage of total execution time

The first three rows show the overhead for sampling based on the DATA_EAR_EVENTS and BRANCH_EVENT events (which count the number of data EAR events and BtrB events respectively). The fourth row shows the overhead for sampling the IP when the CPU_CYCLES event counter overflows. For BRANCH_EVENT and CPU_CYCLES, sampling every 500K events incurs only 1%-2% overhead. For data cache misses, we found that sampling 1 in 50K misses successfully identified hot loads just as well as sampling more frequently while incurring less than 1% overhead.

Since the .TB3 file contains all the samples taken during a monitoring run, we use a post-processing utility called *sfdump* to extract and summarize the necessary information into a text format called the MDB file (Monitor Data Base). In the MDB file, all samples with the same IP are stored in a single combined entry resulting in a 10-100x reduction in file size (versus the .TB3 file).

2.3 The Annotations Library and a Compiler Usage Methodology

Since all the data gathered by the EARs is indexed by IP, the compiler needs a mechanism to map instructions in its internal intermediate representation to the event data saved in the MDB file. To do this, the compiler uses an annotation system to save information such as program control flow graphs, basic block attributes, and instruction level line/column numbers into the binary (much like debug information). Later, the compiler uses an annotations library to read the annotations back out of the binary (by IP) and maps it to PMU data from the MDB file. The annotations library is open source and available for download [Int01].

Although the annotations library can accommodate nearly any structural organization that a compiler might want to emit, we have chosen a convention that allows tools to traverse the annotation hierarchy without knowing all the possible attributes that might be contained in a given binary. These conventions allow new annotations to be emitted by a compiler without having to rewrite existing tools (forward and backward compatibility). The conventions define a program hierarchy of module – file – function – basic block – instruction. At each level, the compiler describes what attributes hang off of each object as well as the parent/child relationship with the other levels (e.g., basic blocks are children of functions).

To build the entire annotation structure, the compiler recursively traverses its internal program representation and creates corresponding objects in the annotation hierarchy and fills in their attribute fields. Since it is likely that not every attribute of every object needs to be given a value, the annotations library optimizes space by compressing unused attribute fields. Once the compiler has completed annotating the binary, they are emitted into the assembly file. Although the annotations increase the physical size of the binary, the linker is directed to store annotation information into a separate section in the binary so that it will not be loaded at run-time.

[1] Similar tools are available for other platforms, such as pfmon for Linux and Caliper for HPUX.

Figure 2 shows the conceptual flow of an annotation-based performance analysis infrastructure. Following the middle column, an application is compiled with the annotations library and the resulting binary is run on hardware with *psamp* gathering PMU data. The resulting .TB3 file is then converted to MDB format. At that point, we have two possible usage models. In both flows, the annotated binary is read and the annotations converted to a text format that is easy to read/parse.

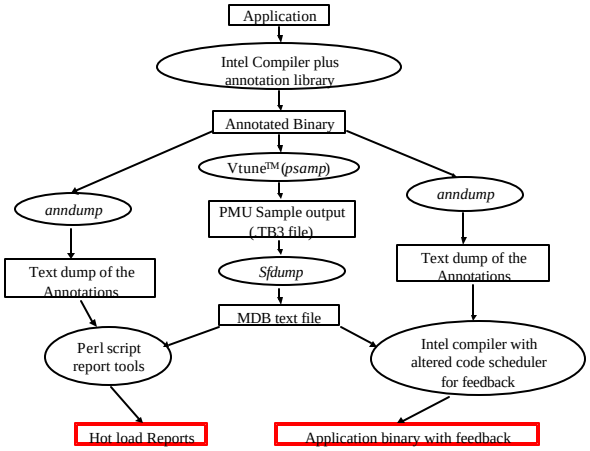

Figure 2 Conceptual annotation based infrastructure

In the left flow, we use the MDB file and the text version of the annotations extracted from the binary to feed an analysis tool that creates run-time reports. By combining the annotation and PMU data, these reports can show what events happened and where (function, block, instruction). Since the MDB and annotations are both available in text format, the tools can be written in a shell language such as Perl.

In the right side flow, the annotated binary is again converted to text, but this time, the MDB and annotation text files are fed back into the compiler, which recompiles the application using the PMU data to enhance its heuristics.

For both flows in Figure 2, Figure 3 shows the specifics of how the MDB and annotations data correlate the dynamic run-time behavior with the static compile-time information. Since the annotations contain the function name, source line and column number, and the IP in the binary, we are able to map the IP event data to an instruction's intermediate representation in the compiler or just use them to print a runtime report for performance analysis.

The infrastructure can handle any IP-specific event data that are generated by the Itanium®2 PMU, although we specifically focused on the data EAR for our case study in the next section.

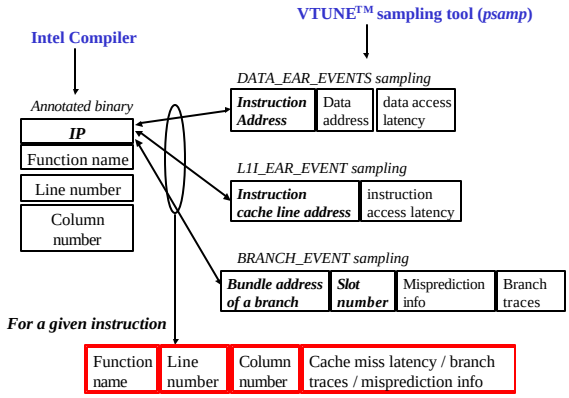

Figure 3 Correlating compile-time information with PMU results

3 Case Study: Reducing Memory Access Stalls

To test our infrastructure, we wanted to attack a problem that has not been effectively handled using current production techniques. Since memory access stalls are one of the largest components of runtime in many applications, we wanted to use PMU features to help reduce this component. Although the compiler already emits explicit prefetches for some predictable data access patterns (e.g., array or linked-list access with fixed strides), it generally doesn't know which loads are missing in cache when it decides to use a prefetch. The fact that we can easily gather this information in the PMU motivated this study more than having specific new prefetching heuristics/ideas in mind.

Additionally, we wanted to demonstrate results on a real system with all their practical limitations and restrictions. Unfortunately, unlike a project where one is experimenting with 'what if' scenarios (what if the processor PMU had this capability, what if the compiler didn't have this type of limitations, etc.) this made it very difficult to try ideas that depending on fixing new-found limitations of our system. On the other hand, all the results shown *demonstrate* lower bounds on what can be achieved. The process of implementing the infrastructure on near-production systems also provides practical learnings on what will be required to productize our techniques. In Section 3.2, we describe our heuristics. In Section 3.3, we analyze the results of applying these heuristics and running the resulting binaries on hardware. In Section 3.4, we discuss the high level learnings from the case study.

3.1 Experimental Setup

Our experiments were run on a single pre-production Intel® Itanium®2 processor using the Intel 870 chipset and 1 GB of main memory. The Itanium®2 processor can issue a maximum of six instructions per clock, composed of up to two integer loads, two stores, six arithmetic operations, three branches, and two floating-point operations. The first level instruction and data caches are 16 KB each, 4-way set associative. The second level cache is 256KB unified, 8-way set associative and the third level cache is 3MB, 12-way set associative. A complete description of the Itanium® 2 processor micro-architecture is available in [Int02].

For our experiments, we selected the SPEC CPU2000 integer benchmarks[Spe00] and a few benchmarks from the Olden benchmark suite[Old96] that are known to be data intensive. The baseline executables were built with instrumentation-based feedback, O2 optimization, and whole-program interprocedural analysis using an internal version of Intel's product Itanium® architecture compiler [IntC02]. [2]

Since the compiler's global code scheduler is a complex decision engine, the only changes we could make that maintained the compiler's scheduling infrastructure were to adjust the scheduling priorities of various instructions, insert new prefetch sequences, and adjust the decision logic that determined whether to speculate a particular instruction. After those choices were made, we let the scheduler continue on its normal path (e.g., considering resource contention, compensation code costs, dependence heights, etc.) to maintain high code quality. Since the Intel compiler has a separate scheduler for software pipelined loops, we turned off software pipelining so that all loop bodies were scheduled using the normal global code scheduler plus our heuristics (turning off software pipelining incurs about a 1% loss on CPU2000 integer codes).

3.2 Feedback Heuristics

We implemented six different scheduling heuristics, each of which can be put into one of two categories, those that insert explicit data prefetch instructions (*lfetch*) and those that try to increase the distance between loads and their dependent instructions. The former helps reduce data cache misses by fetching data in advance of the target load instruction, while the latter attempts to delay use of loaded values that are likely to miss in cache. Table 3 provides a summary of the transformations and scheduling heuristics. The first three rows are the prefetching techniques (H_NEXP, H_GENP, H_BADP), the next 2 rows are scheduling techniques (H_UMUS, H_MAXD), and the last row is a combination of heuristics (H_COMP).

H_NEXP Next line prefetching	To take advantage of spacial locality, prefetch the next 128-byte second-level cache line.
H_GENP General prefetching	To exploit general prefetching, insert a prefetch to the same address as the load, but give the prefetch the highest scheduling priority in the compiler.
H_BADP Base address prefetching	To compute the address of an upcoming load early, take advantage of loads whose addresses are computed as base+constant. Insert a prefetch to the base address and give it very high priority in the scheduler.
H_UMUS Use miss unspeculation	To minimize the stalls related to speculative loads, do not hoist uses of hot loads from their home block. This avoids unnecessary cache miss stalls due to speculative computations whose results might not be needed.
H_MAXD Max load-use distance	To increase the distance between a load and its dependent operations, assign the load a very high priority and dependent instructions a very low priority.
H_COMP Combination prefetching	To combine a few of the heuristics, the compiler selectively applies the combination of the heuristic *H_NEXP, H_BADP, and H_UMUS* depending on the average latency of each load's data cache misses. Temporal locality completers are used to prevent the first level cache from being polluted when the average latency of a load in the *H_NEXP* heuristic is more than 22 cycles.

Table 3 Description of the heuristics and transformations for hot loads

In order to concentrate our heuristics on important loads, we identify those loads that accounted for a significant (either 2 or 6) percent of a program's overall cache cycles as being *hot loads.* For H_NEXP, the threshold was 6%; for all other heuristics, it was 2%. These thresholds were determined by performing test runs to see which values provided the best average performance, although we have not fully investigated complete combinations or individual thresholds for each benchmark.

To implement these heuristics, we altered the compiler's global code scheduler to recognize which loads are hot loads as they became candidates to be scheduled. At that point, we altered either the scheduling priorities (for H_UMUS and H_MAXD) or inserted prefetches (for H_NEXP, H_GENP, H_BADP).

As shown in the example in Figure 4, next line prefetching (H_NEXP) exploits spatial locality for hot loads by prefetching the *next* sequential cache line. The line sizes of Itanium® 2's first and second-level data caches are 64 bytes and 128 bytes, respectively. We chose to prefetch for the second level cache line size since the cache is only five cycles away and experimental results confirmed that it provided better performance on average than prefetching first level cache lines.

[2] Note: the baseline for the Olden benchmarks were not compiled with PGO. Although this impacts the results, our experience is that the percentage of execution time due to data cache stalls is not greatly affected by the optimization level of our current compilers (except for MCF where prefetching is used extensively at high optimization levels).

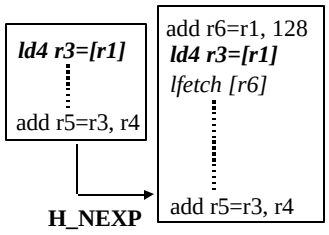

Figure 4 Transformation by heuristic H_NEXP

To try to take advantage of addresses that are available early, general prefetching (H_GENP) inserts lfetches for hot loads and gives the lfetch the highest possible priority for scheduling. Figure 5 shows this transformation. Since both load and lfetch have the same address in this heuristic, the lfetch's scheduling is also bounded by address computation. However, since an lfetch does not change register values and is independent of control and data dependencies, it can be hoisted across calls, branches, and stores without any special support.

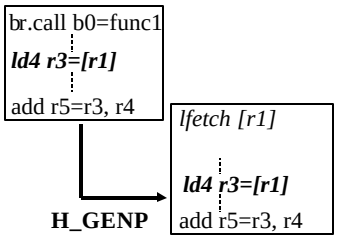

Figure 5 Transformation by heuristic H_GENP

Base address prefetching (H_BADP) searches for loads whose addresses were computed as base+small constant offset and inserts an lfetch for the base address. Figure 6 illustrates this transformation. Since the Itanium architecture does not provide base+offset addressing for memory operations, this transformation guarantees that the prefetch can be scheduled at least one cycle earlier than the load being prefetched.

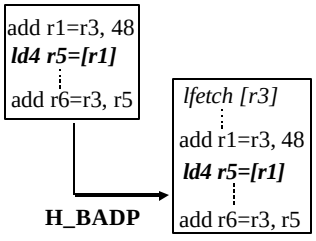

Figure 6 Transformation by heuristic H_BADP

While the first three heuristics tried to improve cache access times by inserting prefetches, the next two heuristics change the way the compiler schedules hot loads and their consumers, but does not insert any prefetch instructions. Use miss unspeculation (H_UMUS) identifies hot speculative loads and attempts to prevent their consumer instructions from being speculated. Figure 7 shows the code before and after speculation. Notice that if the speculated load misses in cache, the *add* instruction will stall waiting for the data, independently of whether the program executes the path from which they were hoisted. For example, if the load misses in cache and then the branch to *label4* were taken, we would have suffered an unnecessary cache-miss-use stall since the results of the load and add are never used.

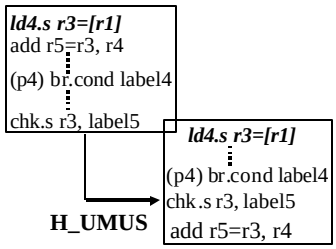

Figure 7 Transformation by heuristic H_UMUS

By forcing the consumers of hot speculative loads to stay in their home blocks, we both increase the distance between a producer and consumer, and we eliminate unnecessary cache-miss use stalls.

As shown in Figure 8, maximize-distance scheduling (H_MAXD) tries to maximize the distance between loads and their consumer instructions. The compiler computes scheduling priority for each instruction and chooses one that has highest priority. By giving highest priority to the loads that miss the data cache and the lowest priority to their consumer instructions, any slack available in the schedule is used to increase the distance between hot loads and their uses. In addition, to further increase distance, we changed the expected latency of hot loads to 10 cycles (the maximum latency to access second-level data cache).

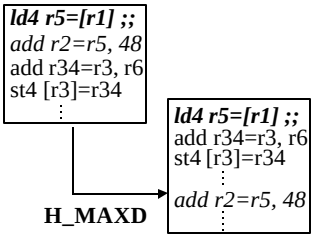

Figure 8 Transformation by heuristic H_MAXD

Finally, to take our learnings from the other approaches, we created a combined heuristic (H_COMP). In this heuristic, we also took advantage of the latency

information provided by the data EAR (on average, how many cycles did each load take to return data) in addition to the basic thresholds described earlier. For H_COMP, we apply H_UMUS in all cases. Additionally, H_COMP tries to apply H_BADP for the loads that miss the data cache and that took 7 or more cycles to return data (experiments showed those with shorter latencies did not benefit). If a given hot load did not qualify for that transformation, then the compiler attempted to apply H_NEXTP if the load showed more than a 12 cycle latency (to bring data in to the second-level cache).

For all of our heuristics, we made several compromises for the sake of completing the case study using available hardware and software. First, we only work on hot loads without respect to their context (cyclic or acyclic regions). This means that our experiments were not able to prefetch across loop iterations. Second, we did not have access to information about address patterns from the high level optimizer to help us make more intelligent choices. Third, we applied the definition of hot loads uniformly, thus ignoring potential benefit from considering criticality and the potential impact of resource constraints. To help counterbalance the resource/criticality issue, we inserted a post-scheduling clean-up phase that removed lfetches if they were scheduled less than two cycles before the load they were targeting. This helps to reduce pressure on the memory system from lfetches with limited potential to reduce latency.

3.3 Results

Figure 9 shows results from SPEC CPU2000 integer benchmarks. Each bar represents the ratio of a heuristic that uses PMU feedback versus the baseline binaries (which are heavily optimized with instrumentation-based feedback, whole program optimization, but no use of PMU data). The first cluster of bars shows relative CPU cycles, the second cluster shows the amount of time spent waiting for loaded data, and the third cluster, the number of first level cache misses.

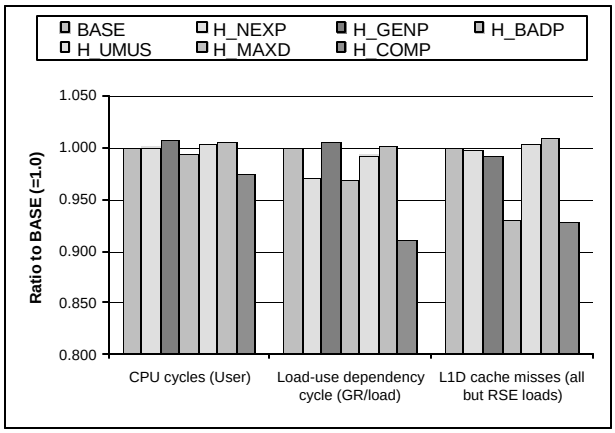

Figure 9 Average results from SPEC CPU2000 integer benchmarks

The combined heuristic *H_COMP* shows the best average speedup (2.5%) with most of the gain coming from mcf (9.4%) and bzip2 (8.1%). H_COMP also reduces average first-level data cache misses by 7% and the average amount of time that instructions are stalled waiting for cache misses by 9%. Next line prefetching (*H_NEXP*) worked especially well for gap (3.9%) and base address prefetching (*H_BADP*) was effective for mcf (9.7%) and bzip2 (2.5%), but the impact on the other benchmarks was not significant.

H_GENP, *H_UMUS*, and *H_MAXD* did not generally improve performance. For H_GENP, the compiler ended up scheduling lfetches too close to the load they were intended to prefetch and were later removed by the cleanup phase. Since H_UMUS and H_MAXD attempt to create distance between a load and its use, this generally results in uses being delayed. While this is beneficial when the load misses in cache, it is harmful when the load hits in cache (and the use could have executed earlier). Thus, we found that these heuristics are only affective for a few spot loads in gzip where the loads miss so frequently that the benefit during cache misses outweighs the losses during cache hits.

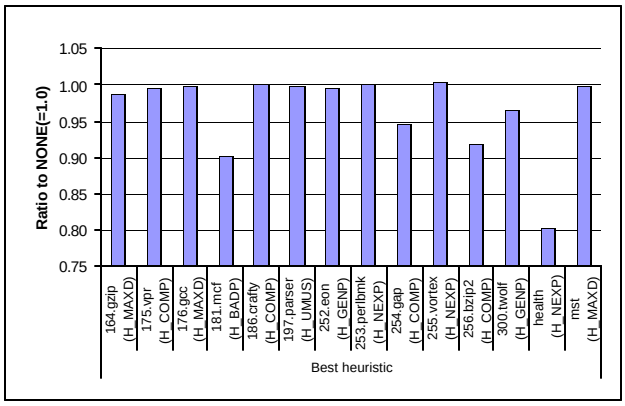

Figure 10 CPU cycle changes by best heuristic for each benchmarks

Figure 10 shows changes in CPU cycles by best heuristic for each benchmark. Each bar is tagged by a benchmark name and the heuristic that worked best for that benchmark. This graph shows that a few benchmarks benefited substantially with our heuristics, but that no one heuristic worked across the board.

Health showed a particularly interesting behavior, where basic next cache line prefetching (*H_NEXP*) reduces CPU cycles by 20%. With further investigation, we found that the single hottest load in the program missed the second-level cache 50% of the time. To avoid perturbing the first level cache, we hand-appended the *.nta* cache placement hint (no temporal locality all levels of the cache) to the lfetch. The .nta hint causes the processor to only bring the line into the second-level cache and to mark it as next to be replaced. This yielded a 46% speed-up due to avoiding cache pollution by unused prefetches. The realization that small changes in heuristics caused great volatility in response was what led to our creating the combined H_COMP heuristic.

Since the results of the heuristics were so difficult to interpret (no consistent winners, small changes making large differences in performance, large changes making little or no difference in performance), we performed an extensive measurement of how well the lfetches that were inserted 'covered' the loads that missed in cache.

Column A	Total number of sampled data cache misses
Column B	The number of cache misses for which the compiler inserted lfetches
Column C	The number of Column B's prefetches were later removed by the compiler
Column D	The final number of cache misses actually covered by remaining lfetches
Column E	The data cache miss coverage (%) by prefetch instructions (= Column D / Column A)

Heuristic	Benchmark	A	B	C	D	E
H_NEXP	164.gzip	25,623	276	0	276	1.08%
H_GENP		25,623	13,666	11,482	2,184	8.52%
H_BADP		25,623	0	0	0	0.00%
H_COMP		25,623	276	0	276	1.08%
H_NEXP	175.vpr	31,093	196	0	196	0.63%
H_GENP		31,093	20,025	15,150	4,875	15.68%
H_BADP		31,093	17,346	573	16,773	53.94%
H_COMP		31,093	196	0	196	0.63%
H_NEXP	176.gcc	4,430	24	0	24	0.54%
H_GENP		4,430	1,360	1,269	91	2.05%
H_BADP		4,430	391	144	247	5.58%
H_COMP		4,430	24	0	24	0.54%
H_NEXP	181.mcf	22,507	14,210	0	14,210	63.14%
H_GENP		22,507	18,554	15,258	3,296	14.64%
H_BADP		22,507	12,846	2,167	10,679	47.45%
H_COMP		22,507	17,626	0	17,626	78.31%
H_NEXP	186.crafty	9,529	0	0	0	0.00%
H_GENP		9,529	226	226	0	0.00%
H_BADP		9,529	0	0	0	0.00%
H_COMP		9,529	0	0	0	0.00%
H_NEXP	197.parser	23,197	0	0	0	0.00%
H_GENP		23,197	8,194	6,559	1,635	7.05%
H_BADP		23,197	4,228	978	3,250	14.01%
H_COMP		23,197	0	0	0	0.00%
H_NEXP	252.eon	33,778	46	0	46	0.14%
H_GENP		33,778	497	497	0	0.00%
H_BADP		33,778	109	60	49	0.15%
H_COMP		33,778	46	0	46	0.14%
H_NEXP	253.perlbmk	7,524	111	0	111	1.48%
H_GENP		7,524	2,219	2,173	46	0.61%
H_BADP		7,524	1,594	67	1,527	20.30%
H_COMP		7,524	1,423	0	1,423	18.91%
H_NEXP	254.gap	4,307	1,479	0	1,479	34.34%
H_GENP		4,307	2,812	1,298	1,514	35.15%
H_BADP		4,307	845	27	818	18.99%
H_COMP		4,307	1,479	0	1,479	34.34%
H_NEXP	255.vortex	8,001	24	0	24	0.30%
H_GENP		8,001	4,272	3,466	806	10.07%
H_BADP		8,001	1,830	741	1,089	13.61%
H_COMP		8,001	638	0	638	7.97%
H_NEXP	256.bzip2	18,142	1,780	0	1,780	9.81%
H_GENP		18,142	16,504	16,295	209	1.15%
H_BADP		18,142	10,941	0	10,941	60.31%
H_COMP		18,142	12,721	0	12,721	70.12%
H_NEXP	300.twolf	36,329	4,852	0	4,852	13.36%
H_GENP		36,329	12,275	7,564	4,711	12.97%
H_BADP		36,329	3,794	797	2,997	8.25%
H_COMP		36,329	4,852	0	4,852	13.36%
H_NEXP	Sum	224,460	22,998	0	22,998	10.25%
H_GENP		224,460	100,604	82,043	18,561	8.27%
H_BADP		224,460	53,924	4,909	49,015	21.84%
H_COMP		224,460	39,281	0	39,281	17.50%

Table 4 Data cache miss coverage by *lfetches* per benchmark

In Table 4, Column A shows the total number data cache miss samples were gathered, and Column B the number for which we inserted lfetches. Column C shows how many were removed because they were eventually scheduled in the same cycle as their target load and Columns D and E show the number and percent of misses that were finally covered at runtime by the inserted lfetch instructions. Note that 'coverage' implies that we

successfully inserted a prefetch, but does not indicate how much of the latency was covered.

Although the heuristic *H_COMP* was our best performer, it covered only 17.5% of total data cache misses. In the case of heuristic *H_GENP*, the initial lfetches inserted covered almost 50% of total data cache misses. However, after the clean-up phase, only 8% of misses were still covered. These results indicate that our detection mechanism (the data EARs, sampling levels, and thresholds used to classify loads as hot or not) are effective at identifying important loads, but that our scheduling/prefetching algorithms are not sufficiently general to handle all the situations.

3.4 Discussion

Although we had some success inserting prefetches, we found that many loads were difficult to optimize because of three basic problems: they were in a short code sequences (e.g., tight loop bodies), they were closely paired with their dependent instructions, or their address computation was on the critical path and lfetches could not be inserted early enough to be effective. The learning is that future efforts need to enable feedback to the loop optimizer to handle cross-iteration scheduling and software pipelining to increase miss coverage.

Our second major learning was that prefetching for locality is easier than prefetching for latency (even with hot load information). We found that we could effectively prefetch second-level cache misses with H_NEXP, but prefetching for loads that hit in the first or second-level cache were far less effective. Part of this is due to the limitations of our scheduler, but part of it is also the inherent criticality of address computations.

The third learning came from the product compiler team's experience and relates to our results. On early steppings of the Itanium® 2, the product compiler team had difficulty effectively using prefetches on floating-point codes. It was discovered that there were limitations in the processor's ability to handle overlapping (usually redundant) outstanding prefetches. When later processor steppings came out that improved the processors' ability to handle overlapping prefetches, performance of the product compiler's prefetching improved dramatically. This shows that relatively small differences in prefetching hardware greatly affect software's ability to use the feature. In this case, the hardware's ability to dispose of overlapping prefetch instructions dramatically improved the usability of the feature for the compiler. The results from our experiments were performed on early steppings of the Itanium® 2 processor and may thus have further room for improvement.

The last major learning is that everything has to be designed to work together from the ground up. We saw that no one heuristic can be used for all benchmarks or even within an individual benchmark. We also found that the scheduler and high-level analysis components of the compiler all need to be cooperating and prepared to use the information that our PMU infrastructure provides. Finally, we saw that even with the hardware prefetch instructions, seemingly small behavioral characteristics greatly impacted the compiler's ability to use prefetches.

4 Related Work

To increase program performance, compilers, binary optimizers and other tools have incorporated dynamic information to bridge the gap between global static information in the compiler and local dynamic information from the processor. In [Smi00], Smith provides a comprehensive review of the challenges present in utilizing feedback-directed optimizations.

Traditionally, tools such as compilers, profilers, and post-link binary optimizers [LS95][Smi91][Rat98][SE94] instrument the code to create a dynamic profile that is then fed to an extra round of optimization [BL94][BL00][CL99][Hwu+93]. Some approaches use hot path information to optimize code layout [DB00]. Spike [CGL+97] uses sampled IP data to reconstruct hot paths and reorders code to improve instruction cache performance. Morph [ZWG+97] is an OS supported profile-directed optimization framework to seamlessly collect and maintain profiles that are used to transform an intermediate representation of the program into a new executable. Dynamo [BDB00] is a software-based on-the-fly dynamic optimization engine used to transparently gather profile information and optimize HP PA-8000 binaries.

With the advent of performance counters maintained by hardware, microarchitectural effects such as caching and branch prediction that were once hidden from external tools can now be used to increase performance. ProfileMe [DHW+96] samples instructions and provides highly detailed information by following individual instructions through the entire pipeline. Event-based sampling in the Pentium 4 [Spr02] allows events to be tracked to exact IPs. Other approaches have tried to adapt existing structures to provide profile data [CPC94].

Focusing on the desire to integrate profile collection and optimization, entire dynamic optimization frameworks have been proposed and evaluated. DCPI [And+97] is a low overhead hardware-based continuous profiling system that samples program counters. The system buffers multiple samples in the hardware using hash tables before invoking an interrupt, thus amortizing the cost of the interrupt over multiple samples. The Compaq C compiler [CL99] also performs many code layout optimizations based on data collected with DCPI. Similar to our experimental framework, HP Caliper [Hun00] takes advantage of the Itanium®2 PMU to analyze applications' performance and provides user APIs for dynamic as well as static binary instrumentation.

The desire to increase communication between static environments and hardware has led to proposals for dynamic optimization in hardware. Replay [PL01] and Daisy [EA97] both provide a microarchitectural framework to support dynamic optimization. The hardware is given the role of constructing hot code sequences, performing optimizations, and recovering from errors. Merten et. al. [MTB+01] propose another such framework based on the IMPACT [ACM+98] compiler.

5 Conclusions

In this paper, we have demonstrated the ability to enhance instrumentation-oriented feedback approaches with hardware-based solutions by enhancing existing infrastructure to make use of new hardware and software usage models. Together with results from previous studies, our work shows that PMU-based statistical sampling can completely eliminate the need for code instrumentation while simultaneously increasing ease of use, speed of profiling, and the level of information that can be gathered. To make this possible, performance monitoring hardware such as that provided by the Itanium® 2 processor event address registers must be available and reasonably efficient.

In our case study, we learned that having the information is not the same as using the information. In our data cache study, we were able to improve scheduling heuristics and insert prefetch instructions targeted at loads known to miss in cache. The results showed that individual heuristics were effective at improving specific benchmarks, but that no one heuristic worked across all benchmarks or even all locations within a single benchmark. These results demonstrated that the entire optimization framework needs to be well-engineered with great attention to every component to effectively make use of the data – hardware, driver, scheduler, software pipeliner, and high-level optimizer.

While our study only concentrated on data cache access, our infrastructure supports the feedback of any IP information the Itanium® 2 processor can deliver, including instruction stream, branch prediction, TLB, and ALAT event information. Now that the infrastructure for testing new compilation heuristics is available, we expect this information to drive the next generation of optimizations to make better use of the many advanced features in the Itanium architecture.

While this paper was targeted at static compilation, efforts are already underway to leverage these techniques for use in managed runtime systems such as Java or .NET. In these systems, low-overhead, high-detail profiling will allow for universal recompilation/optimization of application codes universal while being invisible to the end user. As an intermediate step, another possibility is to productize 'self-monitoring' software that can self-select different code sequences based on real-time PMU feedback. The Itanium®2 processor's PMU and the work presented in this paper are just the beginning steps in enabling the continuous low-overhead collection of detailed microarchitectural behavior to improve application performance.

6 References

[ACM+98] D.I. August, D.A. Connors, S.A. Mahlke, J.W. Sias, K.M. Crozier, B. Cheng, P.R. Eaton, Q.B. Olaniran, and W.W. Hwu. "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture," *Proc. 25th Int'l Symp. on Computer Architecture*, June 1998.

[And+97] J. Anderson, et al. "Continuous Profiling: Where have all the cycles gone?," *Proc. 16th ACM Symposium on Operating Systems Principles*, October 1997.

[BDB00] V. Bala, E. Duesterwald, and S. Banerjia. "Dynamo: A Transparent Runtime Optimization System," *Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[BL94] T. Ball and J. Larus. "Optimally profiling and tracing programs," *ACM Transactions on Programming Languages and Systems, Vol. 16, No. 4*, July 1994.

[BL00] T. Ball and J. Larus. "Efficient Path Profiling," *Proc. 29th Annual Intl. Symp. on Microarchitecture*, June 2000.

[BN00] J. Bharadwaj and R. Narayanaswamy. "Continuous Compilation," *Unpublished Report*, October 2000.

[CGL+97] R. Cohn, D. Goodwin, G. Lowney, and N. Rubin. "Spike: An Optimizer for Alpha/NT Executables," *USENIX Windows NT Workshop*, August 1997.

[CL99] R.Cohn and G. Lowney. "Feedback Directed Optimization in Compaq's Compilation Tools for Alpha," *Proc. 2nd ACM Workshop on Feedback-Directed Optimization*, November 1999.

[CPC94] T. M. Conte, B. A. Patel, J. S. Cox. "Using Branch Handling Hardware to Support Profile-Driven Optimization," *Proc. 27th Annual Intl. Symp. On Microachitecture*, November 1994.

[DB00] E. Duesterwald and V. Bala. "Software Profiling for Hot Path Prediction: Less is More," *Proc. 9th Annual Intl. Conf. On Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 2000.

[DHW+96] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," *Proc. 30th Annual Intl. Symp. on Microarchitecture*, December 1997.

[EA97] K. Ebocioglu and E. Altman. "DAISY: Dynamic Compilation for 100% Architectural Compatibility," *Proc. 24th Annual Intl. Symp. on Computer Architecture*, June 1997.

[EAG+01] K. Ebocioglu, E. Altman, M. Gschwind, and S. Sathaye. "Dynamic Binary Translation and Optimization," *IEEE Trans. On Computers, Vol. 50,*

No. 6, June 2001.

[Hun00] R. Hundt. "HP Caliper: A Framework for Performance Analysis Tools," *IEEE Concurrency, Vol. 8, No. 4*, October-December 2000.

[Hwu+93] W.W. Hwu et. al. "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing*, January 1993.

[Int01] Intel Corporation. Flexible Annotations API Programmer's Guide, May 2001, *Available from http://developer.intel.com/software/products/opensource/tools1/perftools.htm*.

[Int02] Intel Corporation. Intel® Itanium®2 Processor Reference Manual for Software Development and Optimization. June 2002. *Available from http://developer.intel.com/design/itanium2/manuals/index.htm*.

[IntC02] Intel Corporation. Intel Performance Compilers, *Available from http://developer.intel.com/software/products/compilers*.

[IntV02] Intel Corporation. Intel Vtune™ Performance Analyzer, *Available from http://developer.intel.com/software/products/vtune*.

[LS95] J. Larus and E. Schnarr, "EEL: Machine Independent Executable." In Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation, ACM, pages 291-300, June 1995.

[MTB+01] M.C. Merten, A.R. Trick, R.D. Barnes, E.M. Nystrom, C.N. George, J.C. Gyllenhaal, and W.W. Hwu. "An Architecture Framework for Runtime Optimization," *IEEE Trans. On Computers, Vol. 50, No. 6*, June 2001.

[Old96] The Olden benchmark suite, June 1996, *Available from http://www.cs.princeton.edu/~mcc/olden.html.*

[PL01] S.J. Patel and S.S. Lumetta. "rePLay: A Hardware Framework for Dynamic Optimization," *IEEE Trans. On Computers, Vol. 50, No. 6*, June 2001.

[Rat98] Rational Software Corporation. Purify, 1998. http://www.pure.com/products/purify.

[SE94] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools." In Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, ACM, pages 196-205, June 1994. See also Research Report 94/2, Western Research Laboratory, Digital Equipment Corporation.

[Smi91] M. Smith, "Tracing with Pixie." Technical Report CSL-TR-91-497, Computer Systems Laboratory, Stanford University, Stanford, CA, USA, November 1991.

[Smi00] M. D. Smith. "Overcoming the Challenges to Feedback-Directed Optimization," *Proc. ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, January 2000.

[Spe00] SPEC CPU2000 benchmark, December 2000, *Available from http://www.spec.org/osg/cpu2000/*.

[Spr02] B. Sprunt. "Pentium4 Performance Monitoring Features," *IEEE MICRO, Vol. 22, No. 4*, July-August 2002.

[ZWG+97] X. Zhang, Z. Wang, N. Gloy, J.B. Chen, and M.D. Smith. "System Support for Automatic Profiling and Optimization," *Proc. 16th ACM Symposium on Operating Systems Principles*, October 1997.